Safety-Liveness Exclusion in Distributed Computing

Victor Bushkov
EPFL, IC, LPD
victor.bushkov@epfl.ch

Rachid Guerraoui
EPFL, IC, LPD
rachid.guerraoui@epfl.ch

Abstract

The history of distributed computing is full of trade-offs between safety and liveness. For instance, one of the most celebrated results in the field, namely the impossibility of consensus in an asynchronous system basically says that we cannot devise an algorithm that deterministically ensures consensus agreement and validity (i.e., safety) on the one hand, and consensus wait-freedom (i.e., liveness) on the other hand.

The motivation of this work is to study the extent to which safety and liveness properties inherently exclude each other. More specifically, we ask, given any safety property S, whether we can determine the strongest (resp. weakest) liveness property that can (resp. cannot) be achieved with S. We show that, maybe surprisingly, the answers to these safety-liveness exclusion questions are in general negative. This has several ramifications in various distributed computing contexts. In the context of consensus for example, this means that it is impossible to determine the strongest (resp. the weakest) liveness property that can (resp. cannot) be ensured with linearizability.

However, we present a way to circumvent these impossibilities and answer positively the safety-liveness question by considering a restricted form of liveness. We consider a definition that gathers generalized forms of obstruction-freedom and lock-freedom while enabling to determine the strongest (resp. weakest) liveness property that can (resp. cannot) be implemented in the context of consensus and transactional memory.

1 Introduction

The correctness of a distributed algorithm is expressed through *safety* and *liveness* properties [30, 1, 33, 29]. These properties can be defined as sets of histories (traces). More specifically, a safety property is defined as a prefix-closed and limit-closed set of well-formed histories [29]. Whereas, a liveness property is defined as a set of histories that permits any finite well-formed history, i.e., for every finite history there exists a continuation of that history in the liveness property [29].

Because a liveness property basically states that certain 'good' events should eventually happen while a safety property states that certain 'bad' events should never happen, it occurs sometimes that a safety property makes it impossible to guarantee a liveness property, i.e., when the 'bad' and the 'good' coincide. For example, it is impossible to implement a consensus shared object using only asynchronous read/write shared memory while ensuring both wait-freedom, a liveness property, and agreement and validity, a safety property [8, 28]. The history of distributed computing is full of such trade-offs [7, 19, 37, 22, 13, 14, 6, 12, 4].

We ask the questions of *safety-liveness exclusions*: namely if we can determine, given any safety property S, the strongest (resp. weakest) liveness property that can (resp. cannot) be achieved with S. In the context of consensus for instance, this would mean determining the strongest liveness property that can be ensured while still ensuring consensus safety. Given that consensus is at the heart of state machine replication [26, 32, 11, 27], addressing such questions could enable us to determine the maximum availability (liveness) to be expected if the consistency (safety) of a shared distributed service needs to be preserved. More specifically, given $\mathbb{S}$ and $\mathbb{L}$ the sets of safety and liveness properties respectively, we seek to determine if there is a mapping $f_w : \mathbb{S} \to \mathbb{L}$ (resp. $f_s : \mathbb{S} \to \mathbb{L}$) that maps every safety property $S \in \mathbb{S}$ onto the weakest (resp. strongest) liveness property $L \in \mathbb{L}$ that cannot (resp. can) be achieved with S.

When the strongest possible liveness property in a given context is known to be implementable with a given safety property, the answer to the above question is trivial. In many cases, however, the question is more challenging. We show in this paper that, maybe surprisingly, in the general case, it is impossible to determine the strongest (resp. weakest) liveness property that can (resp. cannot) be achieved with a given (non-trivial) safety property. In other words, the safety-liveness exclusion mappings do not exist.

We proceed as follows. We consider a general asynchronous shared-memory distributed system, in which any number of processes can fail by crashing, and we say that a liveness property *excludes* a safety property of a shared object if *every* implementation of that object which ensures the safety property, violates the liveness property. We then show that there is no strongest liveness property that does not exclude a given safety property in a non-trivial case, and give a necessary and sufficient condition when there is a weakest liveness property that excludes a given safety property.

- If L_{max} is the strongest possible liveness property among all liveness properties (L_{max} is a liveness property that requires progress for *all* non-crashed processes), then the first result states the following. If there exists a strongest liveness property that does not exclude a given safety property, then such property must be L_{max}.

- The fact that a liveness property L excludes a safety property S means that there is an adversary w.r.t. L, i.e., an entity that plays against an implementation ensuring S and that decides on the schedule and inputs of processes to win the game by having the implementation violate L. Our second necessary and sufficient condition states that there is a weakest liveness property that excludes safety property S iff for all adversaries w.r.t. L_{max}, the intersection of their behavior is also a behavior of an adversary w.r.t. L_{max}.

For all common safety properties studied in the literature and all common liveness properties, it is possible to find two adversaries the intersection of which behavior is not a behavior of an adversary, e.g. by considering two adversaries that make processes invoke operations with different arguments or different process identifiers, which results in two disjoint behaviors. Thus, for all common safety properties which are impossible to implement together with a corresponding strongest liveness property L_{max}, we have two impossibility results.

They basically state that there are no safety-liveness exclusion mappings.

A corollary of our impossibilities is, for instance, that there is no strongest liveness property that can be implemented (resp. no weakest liveness property which is impossible to implement) together with consensus safety, i.e., agreement and validity. Also, we can now answer questions such as the one raised in [4], about the existence of a weakest transactional memory (TM) liveness property which is impossible to implement together with opacity [15] and the existence of a strongest TM liveness property implementable with opacity*. We show that there are no such liveness properties. In fact, our impossibilities can be applied to many other contexts, such as k-set agreement [3] or high-level object implementations from registers [19] to show that there is no strongest implementable liveness property (resp. no weakest non-implementable liveness property).

One way to circumvent our impossibilities is to restrict the definitions of liveness and safety, i.e., basically to consider subsets $\mathscr{S} \subseteq \mathbb{S}$ and $\mathscr{L} \subseteq \mathbb{L}$, so that it is possible to find mappings $f_w : \mathbb{S} \to \mathscr{L}$ and $f_s : \mathbb{S} \to \mathscr{L}$. We explore a restricted definition of liveness that covers properties of *non-blocking* systems and includes properties that require either *minimal* or *maximal* progress and which are either *dependent* or *independent* of a scheduler.† Our restricted definition combines the notions of *k-obstruction-freedom* [35] and *l-lock-freedom*. While k-obstruction-freedom is a dependent maximal progress guarantee that requires progress of every process in a group of less than k processes which execute alone, l-lock freedom is an independent minimal progress guarantee that requires progress of at least l correct processes. We define a liveness property as a union of l-lock-freedom and k-obstruction-freedom, where $l \leq k$, and call such a property (l,k)*-freedom*. Being general enough, (l,k)-freedom covers liveness properties of shared objects considered most commonly used in literature and allows us to positively answer the safety-liveness exclusion question for many common safety properties. For example, considering consensus in a read-write shared memory system of n processes, the strongest implementable liveness property is $(1,1)$-freedom and the weakest non-implementable is $(1,2)$-freedom; while for TM implementations the strongest implementable liveness property is $(1,n)$-freedom and the weakest non-implementable is $(2,2)$-freedom. Finally, we show that (l,k)-freedom has its own limitations by giving an example of a TM safety property for which we cannot find a weakest liveness that excludes the safety property.

The rest of the paper is organized as follows. Section 2 recalls basic definitions of a shared memory system. Section 3 defines safety and liveness properties. Section 4 poses the safety-liveness exclusion problem. There we give our impossibility results for the general case. Section 5 describes one way to circumvent the impossibility results. Section 6 concludes the paper by discussing alternative ways.

2 System Model

We consider a general shared memory system of n asynchronous *processes* which might stop by crashing. The processes interact with each other only by performing *atomic* primitives on base objects. *Base objects* are shared objects, like read/write registers, test-and-set, compare-and-swap and etc., which are usually provided by the hardware and which are used to implement higher level shared objects. We assume that each process is *sequential* in the sense that after invoking a primitive, the process cannot invoke another primitive until it receives a corresponding response.

A shared object is defined by its type. A shared *object type* $T\,p$ is a tuple (St, Inv, Res, Seq), where St is the set of all possible states of the object, Inv is the set of all possible *invocations* on the object, Res is the set

*It was shown in [4] that in the case of TM shared objects [18, 21, 34], it is impossible to ensure both opacity and local progress, the strongest liveness property. However, the question whether there exists a weakest TM liveness property that excludes opacity or a strongest TM liveness property that does not exclude opacity remained open.

†A *non-blocking* system is a one in which no process p can prevent other processes from making progress once p crashes, i.e. stops participating in a computation. A *maximal* [23] progress guarantee requires progress for all processes while a *minimal* [23] progress guarantee requires progress for some processes. A *dependent* [23] progress guarantee depends on the scheduling of processes while an *independent* [23] progress guarantee has the same requirement irrespectively of how processes are scheduled.

of all possible *responses* from the object, and $Seq \subseteq Inv \times St \times St \times Res$ is the *sequential specification* of the object. An *implementation* I of a shared object of type Tp is a set of algorithms $\{I_1, \ldots, I_n\}$ on base objects such that each algorithm I_i corresponds to a process p_i. When process p_i invokes inv_i on I, where inv is some invocation from Inv, it executes algorithm I_i with inv_i as an input of the algorithm. When executing algorithm I_i process p_i performs *steps* which could be an invocation, a response, or an atomic operation on base objects and local computations. The order in which processes take steps is determined by an external entity called a *scheduler* over which processes have no control. The system is *asynchronous* in the sense that a scheduler may delay any process for an arbitrary long period of time and there is no way for one process to determine if some other process is crashed or delayed.

We use the widely known I/O automata model of asynchronous shared memory systems [29] in which invocations and responses are input and output actions of an automaton. An *I/O automaton* A is a 4-tuple $(states(A), sig(A), init(A), trans(A))$ [29, 33], where:

- $states(A)$ is a (finite or infinite) set of states,
- $init(A) \subseteq states(A)$ is a set of initial states,
- $sig(A) = (in(A), out(A), int(A))$ is an action signature, which partitions the set of all actions $acts(A) = in(A) \cup out(A) \cup int(A)$ into input actions $in(A)$, output actions $out(A)$, and internal actions $int(A)$,
- $trans(A) \subseteq states(A) \times acts(A) \times states(A)$ is a transition relation.

An *execution* of an I/O automaton A is a finite or infinite sequence of alternating states and actions $s_0 \cdot a_1 \cdot s_1 \cdot a_2 \cdot s_2 \cdot \ldots$ such that: (I) $s_0 \in init(A)$, (II) $(s_i, a_{i+1}, s_{i+1}) \in trans(A)$ for every $i \in \{0, 1, 2, \ldots\}$, (III) if the sequence is finite, it should end with a state. The longest subsequence of an execution of A consisting only of actions from $in(A) \cup out(A)$ is called a *history*. An action $a \in acts(A)$ is *enabled* at a state $s \in states(A)$ if there exists a state $s' \in states(A)$ such that $(s, a, s') \in trans(A)$.

Let A_1 and A_2 be two I/O automata. Automata A_1 and A_2 are *compatible* if $out(A_1) \cap out(A_2) = \emptyset$, $int(A_1) \cap acts(A_2) = \emptyset$, and $int(A_2) \cap acts(A_1) = \emptyset$. The *composition* $A_1 \times A_2$ of compatible I/O automata A_1 and A_2 is an I/O automaton A such that:

- $states(A) = states(A_1) \times states(A_2)$,
- $init(A) = init(A_1) \times init(A_2)$
- $sig(A) = (in(A), out(A), int(A))$ is such that:
  - $int(A) = int(A_1) \cup int(A_2) \cup (in(A_1) \cap out(A_2)) \cup (in(A_2) \cap out(A_1))$[‡],
  - $in(A) = (in(A_1) \cup in(A_2)) \setminus int(A)$,
  - $out(A) = (out(A_1) \cup out(A_2)) \setminus int(A)$,
- $((s_1, s_2), a, (s_1', s_2')) \in trans(A)$ iff for every $i \in \{1, 2\}$ the following holds:
  - if $a \in acts(A_i)$, then $(s_i, a, s_i') \in trans(A_i)$,
  - if $a \notin acts(A_i)$, then $s_i = s_i'$.

[‡]Since in our model we assume invocations and responses with unique process identifiers, there is no invocation or response event which is associated with more than one process. Therefore, unlike [29, 33], we use a simplified definition of composition when input and output actions, that are used for communication between components, are hidden by becoming internal actions.

An implementation $I = \{I_1, \ldots, I_n\}$ of a shared object of a type $Tp = (St, Inv, Res, Seq)$ is modeled by the composition $A_I = A_{I_1} \times \ldots \times A_{I_n} \times A_B$. Automaton A_B models the behavior of all base objects used in the implementation (which can be represented as the composition of all automata corresponding to each base object). Each I/O automaton A_{I_i} models the behavior of algorithm I_i such that: $states(A_{I_i})$ is the set of all states of I_i and $init(A_{I_i})$ is the set of initial states of I_i, $\{inv_i | inv \in Inv\} \subseteq in(A_{I_i})$, $\{res_i | res \in Res\} \subseteq out(A_{I_i})$. Output actions from $out(A_{I_i}) \setminus \{inv_i | inv \in Inv\}$ are input actions of base objects and input actions from $in(A_{I_i}) \setminus \{res_i | res \in Res\}$ are output actions of base objects.

A process p_i is *pending in a history* h of A_I if $h|p_i$ ends with an invocation, where $h|p_i$ is the longest subsequence of h consisting only of $acts(A_{I_i})$. Note that $h|p_i$ does not include input and output actions of A_{I_i} which are used for communication with base objects modeled by A_B, this is so since such actions become internal in the composition $A_I = A_{I_1} \times \ldots \times A_{I_n} \times A_B$. Process p_i is *pending at a state* s if s is reachable from an initial state by executing history h in which p_i is pending. An I/O automaton A_I is *input-enabled* if for every process p_i, every invocation $inv \in Inv$, and every state s of A_I the following holds: if process p_i is not pending at state s, then inv_i is enabled at state s. Let h_i be a history consisting only of actions from $\{inv_i | inv \in Inv\} \cup \{res_i | res \in Res\}$, then history h_i is *well-formed*, if h is a sequence of alternating invocations and responses starting with an invocation. A history h of an I/O automaton A_I is *well-formed* if for every A_{I_i} history $h|p_i$ is well-formed. Herein, we consider only well-formed histories and require each I/O automaton A_{I_i} to be input-enabled.

To model process crashes, we augment the model with a special input action $crash_i$ [29] and a special state $s_{crashed,i}$ of A_{I_i} corresponding to each process p_i. At state $s_{crashed,i}$, no action of A_{I_i} is enabled, and from every state $s \neq s_{crashed,i}$ of A_{I_i}, there is a transition $(s, crash_i, s_{crashed,i})$. When an I/O automaton A_{I_i} executes action $crash_i$, it means that process p_i *crashes* and stops executing any steps. Process p_i *crashes in history* h, if h has a $crash_i$ action. Otherwise, process p_i is said to be *correct* in history h. For every object type $Tp = (St, Inv, Res, Seq)$ we denote by $ext_i(Tp)$ the set $(\{inv_i | inv \in Inv\} \cup \{crash_i\} \cup \{res_i | res \in Res\})$ and by $ext(Tp)$ the set $\bigcup_{1 \leq i \leq n} ext_i(Tp)$.

3 Correctness properties

3.1 Safety

A safety property states that some 'bad' events will never happen. For example, *linearizability* [24], *serializability* [31], and *opacity* [15], safety properties of shared memory systems, require that processes never receive responses which result in an inconsistent view of a shared memory system. A safety property is defined as a prefix-closed and limit-closed set of histories [1, 29].

Definition 3.1. *A* safety *property S of a shared object of type $Tp = (St, Inv, Res, Seq)$ is a non-empty set of histories consisting of actions from $ext(Tp)$ such that (1) if $h \in S$, then every prefix of h is in S, and (2) if $h_1, h_2, \ldots$ is an infinite sequence of finite histories in S such that h_j is a prefix of h_{j+1}, for every j, then the unique infinite history which is the limit of $h_1, h_2, \ldots$ is also in S. Implementation I of Tp* ensures *safety property S if every finite history of A_I is in S.*

We assume that each safety property does not contain histories which cannot be implemented, i.e. we assume safety properties S such that for any history $h \in S$, there exists an implementation I such that h is a history of A_I and I ensures S. We also make an additional assumption that a safety property should allow at least one response for any invocation that executes sequentially from an initial state. Specifically, we require that for each $inv \in Inv$ and each process p_i there exists $res \in Res$ such that $inv_i \cdot res_i \in S$.

3.2 Liveness

A liveness property states that some 'good' events will eventually happen. For example, wait-freedom [19, 23] states that every operation by a correct process should eventually return a response. Starvation-freedom [2, 23]

states that every correct process that tries to acquire a lock should eventually succeed. Local progress [4] states that every correct process should eventually commit its transactions.

A liveness property is a property which permits every finite history [30, 1, 29]. For I/O systems, a liveness property states that an input is eventually returned a corresponding output [33]. In particular, for shared memory systems, a liveness property states that some process is eventually returned a desirable response, i.e. *makes progress* [23, 25, 35]. For each shared object type, there is a property that requires progress for all correct processes, the strongest liveness requirement that can be expected. For example, for shared registers, the strongest liveness requirement is wait-freedom. Starvation-freedom is the strongest liveness requirement for lock-based implementations. For TM objects, the strongest liveness requirement is local progress. Hence, for every shared object type we assume a property L_{max} which states the strongest liveness requirement. Formally, with each shared object type $Tp = (St, Inv, Res, Seq)$ we associate some set L_{max} of histories consisting of actions from $ext(Tp)$ such that every finite history h consisting of actions from $ext(Tp)$ is a prefix of some history in L_{max}, i.e. there exists a history h' such that $h \cdot h' \in L_{max}$. Like fairness properties are defined in [38], we define each liveness property as a weakening of L_{max}:

Definition 3.2. *A set L of histories consisting of actions from $ext(Tp)$ is a* liveness *property of a shared object of type Tp if $L_{max} \subseteq L$.*

Let I be an implementation of an object of type Tp. Intuitively, implementation I ensures liveness property L if every fair history of A_I is in L. The restriction to fair histories is necessary because a liveness property cannot require progress from processes which do not get fair turns from a scheduler to perform steps [10]. An execution e of I/O automaton A_I is *fair* if any of the following holds: (I) if e is finite, then no action, other than *crash* actions, is enabled in the final state of e, or (II) if e is infinite, then for every process p_i execution e has either infinitely many actions of A_{I_i} or infinitely many occurrences of states at which no action, other than $crash_i$, of A_{I_i} is enabled. A history h of A_I is *fair* if there exists a fair execution e of A_I such that h is the longest subsequence of e consisting only of input and output actions. Denote by $fair(A_I)$ the set of all fair histories of A_I. Implementation I *ensures* liveness property L if $fair(A_I) \subseteq L$.

The stronger/weaker relation on properties is defined in the standard way [38]. Property L_1 is *weaker* than property L_2 (L_2 is *stronger* than L_1) if every implementation that ensures L_2 also ensures L_1. It easy to see that L_1 is weaker than L_2 iff $L_2 \subseteq L_1$.

4 Safety-liveness Exclusion

A liveness property states that some good events should eventually happen, while a safety property states that some bad events should never happen. Safety and liveness properties exclude each other when the bad events of the safety property coincide with the good events of the liveness property. In the case of shared memory systems, this translates into the impossibility of implementing a shared object that ensures both properties. Let S and L be respectively a safety and a liveness properties of a shared object of type Tp.

Definition 4.1. *Liveness property L* excludes *safety property S if there is no implementation I of an object of type Tp such that I ensures both S and L.*

For example, if Tp is the consensus object type and the set of all possible implementations of Tp is restricted to implementations that use only read-write registers as base objects, then the impossibility result in [8, 5, 28] states that wait-freedom excludes agreement and validity. Another example is the exclusion of local progress and opacity for transactional memory object type [4]. Below we address the following questions: when is it possible to determine a weakest (resp. a strongest) liveness property which excludes (resp. does not exclude) a given safety property? Basically we ask the following question: if $\mathbb{S}$ and $\mathbb{L}$ are the sets of all safety and liveness properties under given general definitions, what is the largest subset $\mathscr{S} \subseteq \mathbb{S}$ for which there exists a map $f_w : \mathscr{S} \to \mathbb{L}$ (resp. $f_s : \mathscr{S} \to \mathbb{L}$) that maps every safety property $S \in \mathscr{S}$ onto the weakest (resp. strongest) liveness property $L \in \mathbb{L}$ that excludes (resp. does not exclude) S.

4.1 Weakest non-implementable liveness

Definition 4.2. *A liveness property L is the* weakest *liveness property to exclude a safety property S if (1) L excludes S and (2) for any liveness property L', if L' excludes S then L' is stronger than L.*

In order to reason about the existence of a weakest liveness property that excludes S, we define a subset of S which we call an adversary set. Informally, an adversary set w.r.t. L and S is a set of histories such that for every implementation I ensuring S there exists a history of I which is in the adversary set and not in L. The existence of an adversary set w.r.t. L and S implies that there is an adversary, which decides on a sequence of steps produced by a scheduler and on invocations sent to implementation I, such that for any implementation the adversary makes I produce an execution that violates L.

Definition 4.3. *An* adversary set *w.r.t. L and S is a non-empty set of histories F such that: (1) $F \subseteq S$, (2) $F \subseteq \overline{L}$, where $\overline{L}$ is the complement of L taken over all well-formed histories, and (3) for every implementation I ensuring S there is a history $h \in fair(A_I)$ such that $h \in F$.*

It is easy to see that L excludes S iff there is an adversary set w.r.t. L and S. The following theorem gives a characterization of safety properties for which it is possible to find a weakest liveness property that excludes them.

Theorem 4.4. *Let $F(L_{max})$ be the set of all adversary sets w.r.t. L_{max} and S, and let $G_{max} = \bigcap_{F \in F(L_{max})} F$. There exists a weakest liveness property that excludes S iff $G_{max} \in F(L_{max})$.*

Proof. Necessary condition: Let L_w be the weakest liveness property that excludes S. Since L_w excludes S, there exists an adversary set F_{L_w} w.r.t. L_w and S.

We first prove that $L_w = \overline{F_{L_w}}$. Assume that $L_w \neq \overline{F_{L_w}}$. By the definition of F_{L_w}, $F_{L_w} \subseteq \overline{L_w}$, and therefore $L_w \subseteq \overline{F_{L_w}}$. Because $L_w \neq \overline{F_{L_w}}$ and $L_w \subseteq \overline{F_{L_w}}$, there is a history h such that $h \in \overline{F_{L_w}}$ and $h \notin L_w$. Because $h \in \overline{F_{L_w}}$ and $L_w \subseteq \overline{F_{L_w}}$, then $L_w \cup \{h\} \subseteq \overline{F_{L_w}}$, and consequently $F_{L_w} \subseteq \overline{L_w \cup \{h\}}$. Consider liveness property $L_w \cup \{h\}$, F_{L_w} is an adversary set w.r.t. $L_w \cup \{h\}$ and S, because $F_{L_w} \subseteq \overline{L_w \cup \{h\}}$. Therefore, $L_w \cup \{h\}$ excludes S and, by the definition of L_w, $L_w \cup \{h\}$ is stronger than L_w, i.e. $L_w \cup \{h\} \subseteq L_w$. This contradicts the fact that $h \notin L_w$.

Next, we prove that for every $F \in F(L_{max})$, $F_{L_w} \subseteq F$. Assume that there is $F \in F(L_{max})$ such that $F_{L_w} \nsubseteq F$. Consider set $\overline{F}$, $\overline{F}$ is a liveness property because $F \subseteq \overline{L_{max}}$ (by the definition of F), and therefore $L_{max} \subseteq \overline{F}$. Since $F \subseteq \overline{\overline{F}}$, then F is an adversary set w.r.t. $\overline{F}$ and S, i.e., $\overline{F}$ excludes S. Since $\overline{F}$ excludes S, $\overline{F}$ is stronger than L_w, i.e. $\overline{F} \subseteq L_w$. Since $F_{L_w} \nsubseteq F$ and $L_w = \overline{F_{L_w}}$, then there exists a history h' such that $h' \notin L_w$ and $h' \in \overline{F}$. This contradicts the fact that $\overline{F} \subseteq L_w$.

Since $F_{L_w} \subseteq F$ for every $F \in F(L_{max})$, then $F_{L_w} \subseteq G_{max}$. By the definition of F_{L_w}, for every implementation I there is $h'' \in fair(A_I)$ such that $h'' \in F_{L_w} \subseteq G_{max}$. Thus, G_{max} is an adversary set w.r.t. L_{max} and S.

Sufficient condition: Because $G_{max} \subseteq \overline{L_{max}}$ (by the definition of an adversary set G_{max}), then $L_{max} \subseteq \overline{G_{max}}$, i.e. $\overline{G_{max}}$ is a liveness property. Since $G_{max} \subseteq \overline{\overline{G_{max}}}$ and $G_{max} \in F(L_{max})$, then G_{max} is an adversary set w.r.t. $\overline{G_{max}}$ and S. Hence, $\overline{G_{max}}$ excludes S.

Consider some liveness property L which excludes S. There is an adversary set F_L w.r.t. L and S. Since $L_{max} \subseteq L$ and $F_L \subseteq \overline{L}$, then $F_L \subseteq \overline{L_{max}}$. Therefore, F_L is an adversary set w.r.t. L_{max} and S. Since G_{max} is the intersection of all adversary sets w.r.t. L_{max} and S, G_{max} is a subset of every adversary set w.r.t. L_{max} and S. From $G_{max} \subseteq F_L \subseteq \overline{L}$ it follows that $L \subseteq \overline{G_{max}}$. Thus, we showed that every liveness property that excludes S is stronger than $\overline{G_{max}}$. By definition, $\overline{G_{max}}$ is the weakest liveness property that excludes S. □

For every object type Tp and most symmetric safety properties S of Tp, i.e. safety properties the requirements of which are the same for all processes irrespectively of process identifiers, it is possible to find two adversary sets F_1 and F_2 w.r.t. L_{max} and S such that $F_1 \cap F_2 = \emptyset$, and consequently $G_{max} \notin F(L_{max})$. Therefore, Theorem 4.4 means that for most safety properties there is no weakest liveness property that excludes the given safety property. We have the following two corollaries:

Corollary: consensus implementations from registers. A consensus shared object is used by processes to agree on some value from a set of proposed values. Each process proposes its own value v by invoking operation $propose(v)$ on a consensus object and receives as a response some value v' (decides value v'). Formally, *agreement and validity*, a safety property of consensus objects, states that all processes decide the same value and the decided value is the value proposed by one of the processes. *Wait-freedom* [19], a liveness property of consensus objects, states that every correct process eventually decides. Consider implementations of consensus that use only read/write registers as base objects. In that case, it is shown in [5] that if two processes propose different values, then there is an execution in which both of them take infinite number of steps and at least one of them does not decide a value. Hence, there exists an adversary set $F_1 = \{propose_1(v) \cdot propose_2(v'), propose_1(v) \cdot v_1 \cdot propose_2(v'), propose_1(v) \cdot propose_2(v') \cdot v_1, propose_1(v) \cdot propose_2(v') \cdot v'_1, propose_1(v) \cdot propose_2(v') \cdot v_2, propose_1(v) \cdot propose_2(v') \cdot v'_2\}$ w.r.t. wait-freedom and agreement and validity, i.e. F_1 is the set of all histories in which two processes propose different value and one of those two processes does not decide. The proof of the impossibility result does not depend on whether process p_1 invokes the *propose* operation first or not, consequently, the set of histories, in which p_2 invokes *propose* before p_1, $F_2 = \{propose_2(v) \cdot propose_1(v'), propose_2(v) \cdot v_2 \cdot propose_1(v'), propose_2(v) \cdot propose_1(v') \cdot v_2, propose_2(v) \cdot propose_1(v') \cdot v'_2, propose_2(v) \cdot propose_1(v') \cdot v_1, propose_2(v) \cdot propose_1(v') \cdot v'_1\}$ is also an adversary set. Since $F_1 \cap F_2 = \emptyset$, it follows that $G_{max} = \emptyset$.

Corollary 4.5. *There is no weakest liveness property of consensus objects which excludes agreement and validity when these objects are implemented only from read/write registers.*

Corollary: transactional memory. Transactional memory (TM) allows to enclose sequential code within atomic transactions which are executed concurrently. Transactional code contains accesses to transactional variables which can be read or written, these variables can be accessed only within transactions. Each transaction is executed only by one process. Processes in a TM implementation can invoke the following transactional operations: *start* which requests to start a new transaction and returns either *ok* or an abort event A, $x.write(v)$ which writes value v to transactional variable x within a transaction and returns either *ok* or an abort event A; $x.read$ which reads a value from transactional variable x within a transaction and returns either a value v or an abort event A; *tryC* which requests to commit a transaction and returns a commit event C or an abort event A. The sequential specification *Seq* of a TM object is such that if a transaction commits, then all the changes made to transactional variables within the transaction are made visible to other processes, and if a transaction aborts, then all the changes are discarded.

Opacity [15], a safety property of TM implementations, states that every transaction, even aborted, observes a consistent state of the system. Specifically, history h ensures opacity if for every finite prefix h' of h there exists a sequential history s such that s is equivalent to some completion $comp(h')$ of h', s preserves the real time order of $comp(h')$, and s respects the sequential specification *Seq*. A *completion* $comp(h)$ of history h is any history derived from h by appending $tryC \cdot A$ for every transaction which does not invoke a commit request in h and by appending either an abort event A or a commit event C for every transaction which invokes a commit request but does not receive a corresponding response in h. Two histories h_1 and h_2 are *equivalent* if for every process p_i, $h_1|p_i = h_2|p_i$. History h_1 *preserves the real time order* of history h_2 if for any two transaction T_1 and T_2 in h_2 if T_1 receives an abort or a commit event, i.e. completes, before T_2 invokes *start* in h_2, then T_1 completes before T_2 starts in h_1. In TM implementations requiring that each operation returns a response is not enough because such requirement can be trivially ensured simply by aborting every transaction. To make progress transactions should be able to eventually commit. Therefore, the set of 'good' events is restricted to commit events. *Local progress* [4], the strongest liveness property of TM implementations, requires that every correct process is eventually given a chance to invoke a commit request $tryC()$ and eventually one of the commit requests $tryC()$ is returned a commit response C. In other words local progress requires that for every correct process eventually there is a transaction which is not aborted. It was shown in [4] that it is impossible to implement a TM object which ensures both opacity and local progress. Consider the adversary set defined by the following strategy [4]:

1. **Step 1.** Process p_1 invokes $start_1$ and waits until it receives as a response ok_1 or A_1. If the response is A_1 the adversary repeats Step 1. Otherwise, process p_1 invokes $x.read_1()$ and waits until it receives as a response v'_1 or A_1. If the response is A_1 the adversary repeats Step 1. Otherwise, the adversary goes to Step 2.

2. **Step 2.** Process p_2 invokes $start_2$ and waits until it receives as a response ok_2 or A_2. If the response is A_2 the adversary repeats Step 2. Otherwise, process p_2 invokes $x.read_2()$ and waits until it receives as a response v''_2 or A_2. If the response is A_2, the adversary repeats Step 2. Otherwise p_2 invokes $x.write_2(v'+1)$, and waits until it receives as a response ok_2 or A_2. If the response is A_2, then the adversary repeats Step 2. Otherwise, p_2 invokes $tryC_2()$ operation and waits until it receives a response C_2 or A_2. If the response is A_2, the adversary repeats Step 2. Otherwise the strategy goes to Step 3.

3. **Step 3.** Process p_1 invokes $x.write_1(v''+1)$ and waits until it receives as a response ok_1 or A_1. If the response is A_1, then the adversary goes to Step 1. Otherwise p_1 invokes $tryC_1()$ operation and waits until it receives a response C_1 or A_1. If the response is A_1, the adversary goes to Step 1. Otherwise the adversary stops.

Let F_1 be the set of all histories produced by the adversary set described by the above strategy, i.e. the set of histories that result from applying the strategy to every possible TM object implementation that ensures opacity. In [4] it is shown that every history which ensures opacity and which is produced by the above strategy violates local progress. Therefore, F_1 is an adversary set w.r.t. local progress and opacity. Let us exchange processes in the above strategy so that process p_1 plays the role of p_2 and vice versa and let F_2 be the set of all possible histories produced by the resulting adversary. Set F_2 is an adversary set w.r.t. local progress and opacity since the impossibility result does not depend on process identifiers. Since local progress is the strongest liveness property of TM objects, then $F_1, F_2 \in F(L_{max})$. However $F_1 \cap F_2 = \emptyset$, because every history from F_1 begins with $start_1$ invocation and every history from F_2 begins with $start_2$. Since $F_1 \cap F_2 = \emptyset$, it follows that $G_{max} = \emptyset$.

Corollary 4.6. *There is no weakest liveness property of TM objects which excludes opacity.*

4.2 Strongest implementable liveness

Definition 4.7. *Liveness property L is the* strongest *liveness property that does not exclude a given safety property S if (1) L does not exclude S, and (2) for any liveness property L' if L' does not exclude S then L' is weaker than L.*

Before proving the theorem which states that L_{max} is the only possible strongest liveness property that does not exclude S in case when such a property exists, we prove the following lemma first.

Lemma 4.8. *The strongest liveness property that an implementation I ensures is $L_{max} \cup fair(A_I)$.*

Proof. By contradiction, assume that there is a liveness property L such that I ensures L and $L_{max} \cup fair(A_I)$ is not stronger than L, i.e., $L_{max} \cup fair(A_I) \nsubseteq L$. Because L is a liveness property, $L_{max} \subseteq L$. And because I ensures L, then $fair(A_I) \subseteq L$. From $L_{max} \subseteq L$ and $fair(A_I) \subseteq L$ it follows that $L_{max} \cup fair(A_I) \subseteq L$. This contradicts the fact that $L_{max} \cup fair(A_I) \nsubseteq L$.

□

Theorem 4.9. *If there is a strongest liveness property that does not exclude S, then it should be L_{max}.*

Proof. Let L_s be the strongest liveness property that does not exclude S. By definition, there is an implementation I_s which ensures both L_s and S. By Lemma 4.8, $L_s = L_{max} \cup fair(A_{I_s})$.

Assume that $L_s \neq L_{max}$, i.e. $fair(A_{I_s}) \setminus L_{max} \neq \emptyset$. We first prove that every history in $fair(A_{I_s}) \setminus L_{max}$ does not include responses. Assume, by contradiction, that there is history $h \in fair(A_{I_s}) \setminus L_{max}$ which includes

a response. Let res_k be the first response in h, i.e. $h = h' \cdot res_k \cdot h''$ and h' does not include any responses. Consider a trivial implementation I_t which does not return responses for any invocation. Let A_{I_t} be an I/O automaton that models I_t. Notice that since the implementation is trivial it does not require any base objects. Because I_t does not return responses, then every history of A_{I_t} consists only of invocations and probably some *crash* events, and therefore, history h is not a history of $fair(A_{I_t})$. By Lemma 4.8, implementation I_t ensures liveness property $L_t = L_{max} \cup fair(A_{I_t})$, note that L_t is not weaker than L_s since $h \notin fair(A_{I_t})$ and $h \in fair(A_{I_s}) \setminus L_{max}$. Let h_t be some history of A_{I_t}. Because h_t consists only of invocations and probably some *crash* events, which are input actions of A_{I_s}, and A_{I_s} is input-enabled, h_t is also a history of A_{I_s}. Because I_s ensures S, $h_t \in S$. Thus, every history of A_{I_t} is in S and therefore I_t ensures S. Hence, L_t does not exclude S. This contradicts the facts that L_s is the strongest liveness property that does not exclude S and L_t is not weaker than L_s.

Next we prove that for any history $h \in fair(A_{I_s}) \setminus L_{max}$, for any process p_l, and for any response res_l, $h \cdot res_l \notin S$ holds. Assume, by contradiction, that there is a history $h \in fair(A_{I_s}) \setminus L_{max}$, a process p_l, a response res_l such that $h \cdot res_l \in S$. Let inv_l be the invocation in h corresponding to response res_l, i.e. $h \cdot res_l | p_l = inv_l \cdot res_l$. Consider history $inv_l \cdot res_l \in S$. Let I be an implementation such that I ensures S and $inv_l \cdot res_l$ is a history of A_I.

Consider a trivial implementation I_b such that: (1) when algorithm I_l^b, corresponding to p_l, receives invocation inv_l the first time it returns res_l, and for the second instance of inv_l it stops without returning any response, (2) when I_l^b receives an invocation other than inv_l it stops without returning any response, and (3) every algorithm I_j^b, where $j \neq l$, once it receives any invocation, it stops without returning any response. Formally, implementation I_b can be described by an I/O automaton $A_{I_b} = A_{I_1^b} \times \ldots \times A_{I_n^b}$ (because I does not use any base objects A_B is omitted from the composition) such that:

- for every $1 \leq i \leq n$, $in(A_{I_i^b}) = \{inv_i | inv \in Inv\} \cup \{crash_i\}$ and $out(A_{I_i^b}) = \{res_i | res \in Res\}$;
- when $A_{I_l^b}$ executes inv_l from an initial state, $A_{I_l^b}$ changes its state to a state s^l at which only $crash_l$ and output action res_l are enabled. Let s^l_{en} be a state of $A_{I_l^b}$ such that there is a transition (s^l, res_l, s^l_{en}) in $A_{I_l^b}$; note that from state s^l_{en} every invocation is enabled. For every invocation $inv' \in Inv$, when $A_{I_l^b}$ executes inv'_l from s^l_{en}, $A_{I_l^b}$ changes its state to a state s^l_1 at which only $crash_l$ action is enabled;
- for every invocation $inv'' \in Inv$, such that $inv'' \neq inv$, when $A_{I_l^b}$ executes inv''_l from an initial state $A_{I_l^b}$ changes its state to a state s^l_2 at which only $crash_l$ action is enabled;
- for every $j \neq l$ and for every invocation $inv' \in Inv$, when $A_{I_j^b}$ executes inv'_j from an initial state, $A_{I_j^b}$ changes its state to a state s^j_1 at which only $crash_j$ action is enabled.

Let h' be any history of A_{I_b} such that h' contains a response. Because I_b returns a response only for the first instance of inv_l, it follows that the response in h' is res_l and $inv_l \cdot res_l$ is a prefix of $h'|p_l$. Because $inv_l \cdot res_l$ is a history of A_I and A_I is input-enabled, then h' is also a history of A_I. Because I ensures S, it follows that every such history h' is in S. Let h'' be any history of A_{I_b} such that h'' does not contain a response. Because A_I is input-enabled, h'' is also a history of A_I and, consequently, $h'' \in S$. Hence, every history of A_{I_b} is in S and, consequently, I_b ensures S. Because (1) $h|p_l = inv_l$, and (2) state s^l of I/O automaton $A_{I_l^b}$ is reachable by executing inv_l from an initial state, and (3) at s^l output action res_l is enabled then every execution of A_{I_b}, to which history h corresponds, is not fair, and consequently $h \notin fair(A_{I_b})$.

By Lemma 4.8, implementation I_b ensures liveness property $L_b = L_{max} \cup fair(A_{I_b})$. Hence, L_b does not exclude S. Because $h \in L_s$ and $h \notin L_b$, liveness property L_b is not weaker than liveness property L_s. This contradicts the facts that L_s is the strongest liveness property that does not exclude S and L_b does not exclude S.

We proved that for every history $h \in fair(A_{I_s}) \setminus L_{max}$ the following holds: (1) h does not contain any responses and (2) there is no process p_i and response res such that $h \cdot res_i \in S$.

Let h be any history in $fair(A_{I_s}) \setminus L_{max}$ and let inv_i be some invocation in h by a process p_i which does not crash in h. Let res_i be a response such that $inv_i \cdot res_i \in S$ and let I' be an implementation which ensures S and which has $inv_i \cdot res_i$ as its history. Because $A_{I'}$, an I/O automata model of I', is input-enabled, it follows that every history h' such that h' includes a single response and $h'|p_i = inv_i \cdot res_i$ is a history of $A_{I'}$. Because I' ensures S, it follows that every such history $h' \in S$. This contradicts the fact that $h \cdot res_i \notin S$. □

Corollary: consensus objects. Consider those implementations of consensus that use only read/write registers as base objects. Since L_{max}, which is wait-freedom, is impossible [5] to ensure together with agreement and validity using only registers, then we have the following corollary:

Corollary 4.10. *There is no strongest liveness property of consensus objects which does not exclude agreement and validity when these objects are implemented only from read/write registers.*

Corollary: transactional memory. Because L_{max}, which is local progress, is impossible [4] to ensure together with opacity, then we have the following corollary:

Corollary 4.11. *There is no strongest liveness property of TM objects which does not exclude opacity.*

5 Circumventing the Impossibilities

In this section we show that if we restrict the space of liveness properties, then it is possible to determine weakest (strongest) liveness property that excludes (does not exclude) a safety property in certain cases.

5.1 (l,k)-Freedom

Instead of giving a general set-theoretic definition of liveness, we give a definition based on the notion of progress [23]. Accordingly, liveness properties can be classified into properties which require either *maximal* or *minimal* progress and which are either *dependent* or *independent* of a scheduler. *Maximal* progress properties require progress for *every* correct process, while minimal progress properties require progress for *some* correct processes. *Wait-freedom*, which requires progress for every correct process, and *obstruction-freedom*, which requires progress only for processes that eventually run without step contention, are examples of maximal progress properties. *Lock-freedom*, which requires progress for at least one correct process, is an example of a minimal progress property. Wait-freedom and lock-freedom are examples of independent progress properties, while obstruction-freedom is an example of dependent progress property. We give a definition of liveness that encompasses both maximal and minimal progress and dependent and independent guarantees.

Intuitively, we say that a process makes progress in an execution if it eventually receives 'good' responses in that execution for its invocations. However, for different object types, the notion of a 'good' response might be different. For example, for objects like consensus or registers, any response is a 'good' response; but for TM objects 'good' responses are those which do not abort transactions. Therefore, we assume that for each object type $Tp = (St, Inv, Res, Seq)$ there is a fixed subset of responses $G_{Tp} \subseteq Res$ which are necessary to make progress. We say that a correct process p_i *makes progress* in a fair execution e if e contains infinitely many responses from $\{res_i | res \in G_{Tp}\}$.

Between the independent progress guarantee of wait-freedom and the dependent progress guarantee of obstruction-freedom lie intermediate maximal progress guarantees that can be satisfied only when up to k processes are scheduled to take steps. Such progress guarantees are grouped into k-obstruction-freedom properties. *k-Obstruction-freedom* [35] states that if at any point in an execution there are at most k processes taking steps, then all these processes should make progress. Likewise, between the maximal progress guarantee of wait-freedom and the minimal progress guarantee of lock-freedom lie intermediate independent

progress guarantees that require progress for at least l processes. We introduce the notion of l-lock-freedom to group such guarantees. *l-Lock-freedom* states that at least l processes should make progress if there are at least l correct processes in an execution or all processes should make progress if there are less than l correct processes in an execution.

l-Lock-freedom and k-obstruction-freedom are different kinds of progress requirements. We combine these two kinds of requirements into a more general one. Specifically, we define (l,k)-*freedom*, where $l \leq k$.

Definition 5.1. *A fair execution e ensures (l,k)-freedom if the following holds. If at most k processes take infinitely many steps in e, then*

- *if at least l processes are correct in e, then at least l processes make progress in e,*
- *if less than l processes are correct in e, then all correct processes make progress in e.*

Observe that if a process is correct in e (i.e. it does not crash) it does not necessary mean that it takes infinitely many steps in e, e.g. a correct process might be prevented from taking steps by the implementation which does not enable steps after certain state. Notice that if LF_l is the set of all executions that ensure l-lock-freedom and OF_k is the set of all executions that ensure k-obstruction-freedom, then $LF_l \cup OF_k$ is the set of all executions that ensure (l,k)-freedom. An implementation I is (l,k)-free if every fair execution of I ensures (l,k)-freedom.

Liveness properties defined as (l,k)-freedom are not totally ordered. For example, consider $(1,3)$-freedom and $(2,2)$-freedom properties. An execution in which only two processes take steps and only one of those two processes makes progress ensures $(1,3)$-freedom but does not ensure $(2,2)$-freedom. An execution in which only three processes take steps and none of those three processes makes progress ensures $(2,2)$-freedom but does not ensure $(1,3)$-freedom. Therefore, $(1,3)$-freedom and $(2,2)$-freedom are incomparable to each other. However, despite the fact that (l,k)-freedom properties are not totally ordered, it is possible to find weakest non-implementable and strongest implementable (l,k)-freedom properties in most cases.

5.2 Consensus and TM examples

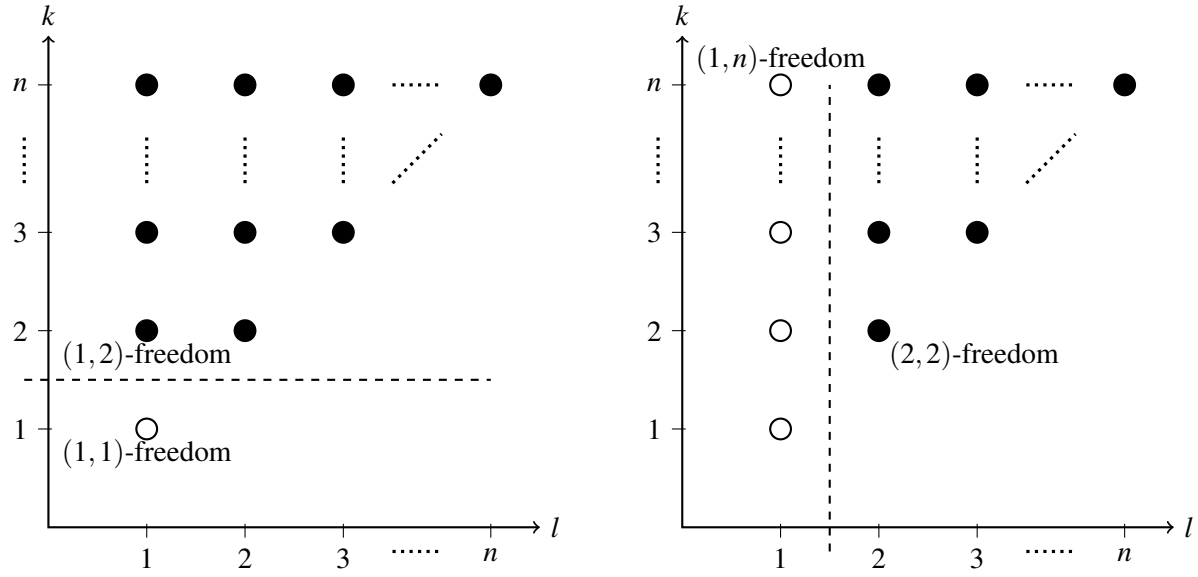
Consider implementations of consensus that use only read/write registers as base objects. The impossibility of consensus from registers [5] says that if two processes invoke two $propose(v)$ operations with different arguments, then there is an adversary set that produces a fair execution in which none of the two processes ever decides a value. Consequently, $(1,2)$-freedom excludes agreement and validity in the case of implementations from registers. Because $(1,1)$-freedom, which is obstruction-freedom, is the only consensus liveness property which is weaker than $(1,2)$-freedom and which is possible [20, 17] to implement using registers, we have the following theorem.

Theorem 5.2. *For implementations of consensus from read/write registers $(1,1)$-freedom is the strongest (l,k)-freedom property that does not exclude agreement and validity and $(1,2)$-freedom is the weakest (l,k)-freedom property that excludes agreement and validity.*

As another example, consider TM shared objects. In [4] it is shown that any non-blocking liveness property which is *biprogressing* is impossible to implement in the TM context with safety properties like *strict serializability* [31] or *opacity* [15] in a system of two processes. A *biprogressing* liveness property is a property which requires progress for at least two correct processes. In the case of (l,k)-freedom properties, the weakest biprogressing property is $(2,2)$-freedom. Since in the TM context it is possible [9] to implement $(1,n)$-freedom, which is the strongest property that requires progress for at most one process, together with opacity, we have the following theorem.

Theorem 5.3. *For TM implementations $(1,n)$-freedom is the strongest (l,k)-freedom property that does not exclude opacity and $(2,2)$-freedom is the weakest (l,k)-freedom property that excludes opacity.*

It is worth noting that in case of (l,k)-freedom liveness properties, the strongest implementable TM liveness property, $(1,n)$-freedom, is not weaker than the weakest non-implementable TM liveness property, $(2,2)$-freedom. In fact, these two properties are incomparable.



(a) S is agreement and validity of consensus.

(b) S is opacity of TM.

Figure 1: (l,k)-Freedom properties can be represented as points on a two-dimensional plane. The more a point located to the right and the higher it is, the stronger the corresponding (l,k)-freedom property is. White points indicate (l,k)-freedom properties that do not exclude a given safety property S, and black ones exclude S.

5.3 Limitations of (l,k)-freedom: a counterexample

Even though (l,k)-freedom allows to circumvent the safety-liveness exclusion impossibility for consensus and TM from Section 4, there are still some safety properties for which it is not the case. For instance, consider the following safety property S of TM. A TM history h ensures S if the following hold:

1. h ensures opacity, and
2. for any three (or more) concurrent transactions $T_1, T_2, T_3, \ldots$ in h, executed by different processes $p_1, p_2, p_3, \ldots$, respectively, if (1) there exists a number t s.t. for each process $p_i \in \{p_1, p_2, p_3, \ldots\}$, T_i is the t-th transactions in $h|p_i$ and (2) each T_i invokes $tryC()_i$ *after* at least other two transactions from $\{T_1, T_2, T_3, \ldots\}$ receive a response for a $start()_j$ operation in h, where $j \neq i$, then $T_1, T_2, T_3, \ldots$ should be aborted in h.

In other words, safety property S requires opacity plus an additional requirement that if three or more transactions have the same timestamp t and invoke a commit request $tryC()$ after any two of them receive a response for starting a transaction request $start()$, then such transactions should be aborted. In [4] it is shown that $(2,2)$-freedom is impossible to implement together with opacity. Because S includes the requirement of opacity, then in the TM context $(2,2)$-freedom excludes S. In order to show that $(1,3)$-freedom excludes S, consider the following adversary:

1. **Step 1.** Processes p_1, p_2, p_3 concurrently invoke $start()$ request to start a transaction and each waits for a response which should be either ok or A. Once each of the three processes receives a response, the adversary goes to Step 2.

2. **Step 2.** Processes which did not receive an abort event A at Step 1, concurrently invoke $tryC()$ request to commit their transaction and each waits for a response which should be either C or A. If each process receives an abort event A, the adversary goes to Step 1. Otherwise the adversary stops.

Assume that the adversary terminates, i.e. there exists a history h such that some process commits in h. Without loss of generality assume this process to be p_1. Let the committing transaction T_1 be a t-th transaction of p_1. According to the adversary, the three processes invoke new transactions concurrently to each other, and consequently the committing transaction is concurrent to the other two t-th transactions of p_2 and p_3. According to the adversary, T_1 invokes $tryC()$ only after p_2 and p_3 receive responses for $start()$ at Step 1 (which eventually happens because of $(1,3)$-freedom). Thus the conditions of the second requirement of property S are satisfied, and consequently, transactions T_1, T_2, T_3 should be aborted - a contradiction. Since the adversary never terminates, none of the three correct processes receives a commit response. Therefore the history produced by the adversary violates $(1,3)$-freedom.

In order to show that there is no weakest (l,k)-freedom property non-implementable with S we show that $(1,2)$-freedom, which is weaker than both $(1,3)$-freedom and $(2,2)$-freedom, is implementable with S. Below we give a simple TM implementation, which is a modification of algorithm A_{GP} from [16], the purpose of which is to show that $(1,2)$-freedom does not exclude S.

The main idea of the implementation is the following. The implementation uses a single shared compare-and-swap object which holds a version number and a value of each transactional variable. Additionally, it uses a shared snapshot object of n registers, where n is the total number of processes. When a process p_i starts a new transaction, it increments its local timestamp, writes the new timestamp to the i-th register, and copies the content of the compare-and-swap object to its local memory. The process performs transactional reads and writes using only the local memory. When the process invokes a commit request it first takes the snapshot of the registers and checks if at least two other processes have a greater timestamp. If so, the transaction is aborted. Otherwise, the process tries to update the compare-and-swap object with a new version number and new values from its local memory. The version numbers are used to ensure opacity, while the timestamps are used to ensure the second requirement of S.

Lemma 5.4. *Algorithm $I_{(1,2)}$ implements a TM that ensures S and $(1,2)$-freedom.*

Proof. Consider any history h of $I_{(1,2)}$. Within this proof, we say that a transaction T_k *reads version w* if T_k is returned in $start_i$ a tuple $(version, oldval)$, where $version = w$. We say that T_k *commits version w* if T_k reads version $w-1$ and is returned value *true* from operation *compare-and-swap* on C in $tryC()$.

The first requirement of S (Opacity). Observe that the version number stored in object C can only increase with time. Hence, if a transaction T_i precedes a transaction T_k in h, then T_i cannot read a version higher than the version read by T_k. Note also that from any set of transactions that read the same version w at most one transaction can commit version $w+1$. Therefore, there exists a total order $\ll$ on the set of transactions in h, such that, for all transactions T_i and T_k in H, $T_i \ll T_k$ if:

- T_i precedes T_k in h; or
- T_i reads a version w_i, T_k reads a version w_k, and $w_i < w_k$; or
- T_i reads a version w_i and T_k commits version $w_i + 1$.

Let h' be the completion of history h such that a transaction T_k in h' is committed in h' iff T_k commits some version in h. Let s be the following sequential TM history:

$$s = h'|T_{\sigma_1} \cdot h'|T_{\sigma_2} \cdot \ldots,$$

where $T_{\sigma_1} \ll T_{\sigma_2} \ll \ldots$ and $h'|T_{\sigma_j}$ is the longest subsequence of h' consisting only of invocations and responses of T_{σ_j}. Clearly, s is equivalent to h' and s preserves the real-time order of h. Every transaction T_k that is

uses: C—compare-and-swap object; R[1, ..., n]—snapshot object (other variables are process-local)
initially: $C = (1,(0,0,\ldots))$, $R[1,\ldots,n] = (0,\ldots,0)$ $version = \perp$, $timestamp = 0$, and $count = 0$ (at every process p_i)

operation $start()_i$
  $timestamp \leftarrow timestamp + 1$
  $R[i] \leftarrow timestamp$
  $(version, oldval) \leftarrow C.read$
  $values \leftarrow oldval$
  return ok_i
end

operation $x_m.read()_i$
  return $values[m]_i$
end

operation $x_m.write(v)_i$
  $values[m] \leftarrow v$
  return ok_i
end

operation $tryC()_i$
  $snapshot \leftarrow R.scan()$
  for $j \leftarrow 1$ **to** n **do**
    if $snapshot[j] \geq timestamp$ **then**
      $count \leftarrow count + 1$
    end
  end
  if $count \geq 3$ **then**
    $count \leftarrow 0$
    return A_i
  end
  $count \leftarrow 0$
  if $C.compare\text{-}and\text{-}swap((version, oldval), (version + 1, values))$ **then**
    $version \leftarrow \perp$
    return C_i
  else
    $version \leftarrow \perp$
    return A_i
  end
end

Algorithm 1: Algorithm $I_{(1,2)}$ implementing a TM that ensures S and $(1,2)$-freedom.

aborted in s does not commit any version, and so T_k does not change the state of base object C. Hence, aborted transactions are effectively invisible to other transactions. Every transaction T_k that is committed in s reads some version w and commits version $w+1$. Since the version number stored in C never decreases, no other transaction commits version $w+1$ in h. transaction T_k thus reads the current snapshot of t-variable values from C, modifies the snapshot locally within operations *read* and *write*, and then atomically changes the state of C from the old snapshot, with version number w, to the new snapshot, with version number $w+1$. Therefore, all transactions that follow T_k in s observe all the values written to t-variables by T_k, and by all preceding transactions that are committed in s. Therefore, history respects the sequential specification *Seq* of TM objects, and so history h ensures opacity. This means that, TM implementation $I_{(1,2)}$ ensures opacity.

The second requirement of S. Observe that each time a process starts a new transaction it increments its timestamp by 1 and announces the new timestamp to other processes by updating the corresponding register $R[i]$. Let h be any history of $I_{(1,2)}$ and let $T_1, T_2, T_3, \ldots$ be any three (or more) concurrent transactions in h, executed by different processes $p_1, p_2, p_3, \ldots$, respectively, s.t. there exists number t s.t. for each $p_i \in \{p_1, p_2, p_3, \ldots\}$, T_i is the t-th transactions in $h|p_i$ and each T_i invokes $tryC_i$ *after* some other two transactions are returned a response for a $start()_j$, where $j \neq i$ and $j \in \{1,2,3\}$, operation in h.

Because each $T_1, T_2, T_3, \ldots$ is the t-th transaction of a corresponding process, when $T_1, T_2, T_3, \ldots$ execute $start()$, the timestamps of each process becomes t and is written to the corresponding register. Since each T_1, T_2, T_3 invokes $tryC()$ after some other two processes get responses for $start()$, each transaction observes the new timestamps of the other two transactions. Therefore, *count* becomes at least 3 during the execution of $tryC()$ of each of the transaction. And consequently, each of the transaction aborts in h.

$(1,2)$**-Freedom.** Consider a fair execution e of $I_{(1,2)}$ and its corresponding history h such that only two processes take infinitely many steps (and therefore only two processes are correct). Consider any correct and live, i.e. transaction which is not aborted or committed, transaction T_k in history h. A transaction T_k can be aborted by $I_{(1,2)}$ only if T_k either it encounters that some other transaction has a higher timestamp tm or T_k reads some version w and then fails to commit version $w+1$. Since eventually there are only two processes that take steps, eventually transactions can be aborted only when T_k reads some version w and then fails to commit version $w+1$; this can happen only if some other transaction by the second correct process commits version $w+1$. Since a transaction that commits any version cannot be aborted, there are infinitely many transactions in h that are not aborted, and so history h, and thus also TM implementation $I_{(1,2)}$, ensures $(1,2)$-freedom. □

6 Concluding Remarks

Our impossibility results mean that if we consider too general definitions of liveness, then it is impossible to answer questions about strongest (weakest) implementable (non-implementable) liveness property. To circumvent these impossibility results it is necessary to consider more restricted definitions of liveness. We have considered one such restriction in the previous section. Here we discuss alternative ones.

For example, we may consider the notion of S-freedom [36], which states that for every set of correct processes P, where S is a set containing some natural numbers and $|P| \in S$, every process in P should make progress as long as it does not encounter step contention with processes outside P. A characterization of liveness properties of consensus objects is given in [36] which partitions liveness properties into implementable (using registers only) and non-implementable properties; S-freedom is then shown to be implementable iff $|S| = 1$. However, even such restricted definition of liveness does not allow to determine a strongest (or a weakest) implementable (or non-implementable) liveness property with a safety property like consensus agreement and validity. Because none of the S-freedom properties with $|S| = 1$ is comparable to each other, there is no strongest S-freedom consensus liveness property implementable using only registers.

One way to circumvent the issue is to consider a restricted definition of liveness which totally orders all liveness properties. For example, the notion of (n,x)-liveness [25] states that x processes should be wait-free and $n-x$ processes should be obstruction-free. It is shown [25] that if $x \geq 1$, then it is impossible to implement a consensus object from registers. Since the set of all (n,x)-liveness properties is totally ordered, then the strongest (and the only) implementable liveness property is $(n,0)$-liveness, and the weakest non-implementable liveness property is $(n,1)$-liveness. Liveness properties can also be defined based on the notion of k-obstruction-freedom [35] which states that if processes from a set P, where $|P| \leq k$, do not encounter step contention with processes outside of P, then every process from P should make progress. According to this definition, the strongest implementable liveness property of a consensus object is 1-obstruction-freedom and the weakest non-implementable is 2-obstruction-freedom. However, the definitions of liveness in [25, 35] do not include some common liveness properties, e.g. lock-freedom that requires progress for at least one process. There is a compromise between how general a definition of liveness should be and the possibility of totally ordering the defined liveness properties.

7 Acknowledgements

We are very grateful to Cheng Wang who discovered the mistake on the example of an adversary set for consensus. We also wish to thank the anonymous reviewers for their helpful comments.

This work has been supported by the European Commission under the 7th Framework Program through the TransForm (FP7-MC-ITN-238639) project and by the European Research Council under the Adversary-Oriented Computing project (ERC-2013-AdG-339539-AOC).

References

[1] B. Alpern and F. B. Schneider. Defining liveness. *Information Processing Letters*, 21(4), 1985.

[2] H. Attiya and J. Welch. *Distributed Computing: Fundamentals, Simulations and Advanced Topics*. John Wiley & Sons, 2004.

[3] E. Borowsky and E. Gafni. Generalized flp impossibility result for t-resilient asynchronous computations. In *ACM STOC*, 1993.

[4] V. Bushkov, R. Guerraoui, and M. Kapałka. On the liveness of transactional memory. In *ACM PODC*, 2012.

[5] B. Chor, A. Israeli, and M. Li. On processor coordination using asynchronous hardware. In *ACM PODC*, 1987.

[6] F. Ellen, P. Fatourou, E. Kosmas, A. Milani, and C. Travers. Universal constructions that ensure disjoint-access parallelism and wait-freedom. In *ACM PODC*, 2012.

[7] F. Fich and E. Ruppert. Hundreds of impossibility results for distributed computing. *Distrib. Comput.*, 16(2-3), 2003.

[8] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2), 1985.

[9] K. Fraser. Practical lock freedom. In *PhD thesis, Cambridge University Computer Laboratory*, 2003.

[10] P. Ganty and R. Majumdar. Algorithmic verification of asynchronous programs. *ACM Trans. Program. Lang. Syst.*, 34(1), 2012.

[11] S. Gilbert and N. Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33(2), 2002.

[12] S. Gilbert and N. A. Lynch. Perspectives on the cap theorem. *Computer*, 45(2), 2012.

[13] D. S. Greenberg, G. Taubenfeld, and D.-W. Wang. Choice coordination with multiple alternatives (preliminary version). In *WDAG*, 1992.

[14] R. Guerraoui and M. Kapalka. On obstruction-free transactions. In *ACM SPAA*, 2008.

[15] R. Guerraoui and M. Kapalka. On the correctness of transactional memory. In *ACM PPoPP*, 2008.

[16] R. Guerraoui and M. Kapalka. *Principles of Transactional Memory*. Morgan and Claypool, 2010.

[17] R. Guerraoui and E. Ruppert. Anonymous and fault-tolerant shared-memory computing. *Distributed Computing*, 20(3):165–177, 2007.

[18] T. Harris, J. R. Larus, and R. Rajwar. *Transactional Memory, 2nd edition*. Morgan and Claypool, 2010.

[19] M. Herlihy. Wait-free synchronization. *ACM Trans. Program. Lang. Syst.*, 13(1), 1991.

[20] M. Herlihy, V. Luchangco, and M. Moir. Obstruction-free synchronization: Double-ended queues as an example. In *IEEE ICDCS*, 2003.

[21] M. Herlihy, V. Luchangco, M. Moir, and W. N. Scherer, III. Software transactional memory for dynamic-sized data structures. In *ACM PODC*, 2003.

[22] M. Herlihy and N. Shavit. The topological structure of asynchronous computability. *J. ACM*, 46(6), 1999.

[23] M. Herlihy and N. Shavit. On the nature of progress. In *OPODIS*, 2011.

[24] M. P. Herlihy and J. M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3), 1990.

[25] D. Imbs, M. Raynal, and G. Taubenfeld. On asymmetric progress conditions. In *ACM PODC*, 2010.

[26] L. Lamport. Using time instead of timeout for fault-tolerant distributed systems. *ACM Trans. Program. Lang. Syst.*, 6(2), 1984.

[27] B. W. Lampson. How to build a highly available system using consensus. In *WDAG*, 1996.

[28] M. C. Loui and H. H. Abu-Amara. *Memory requirements for agreement among unreliable asynchronous processes*, volume 4. JAI press, 1987.

[29] N. A. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers Inc., 1996.

[30] S. Owicki and L. Lamport. Proving liveness properties of concurrent programs. *ACM Trans. Program. Lang. Syst.*, 4(3), 1982.

[31] C. H. Papadimitriou. The serializability of concurrent database updates. *J. ACM*, 26(4), 1979.

[32] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: a tutorial. *ACM Comput. Surv.*, 22(4), 1990.

[33] R. Segala, R. Gawlick, J. Søgaard-Andersen, and N. Lynch. Liveness in timed and untimed systems. *Inf. Comput.*, 141(2), 1998.

[34] N. Shavit and D. Touitou. Software transactional memory. In *ACM PODC*, 1995.

[35] G. Taubenfeld. On the computational power of shared objects. In *OPODIS*, 2009.

[36] G. Taubenfeld. The computational structure of progress conditions. In *DISC*, 2010.

[37] G. Taubenfeld and S. Moran. Possibility and impossibility results in a shared memory environment. *Acta Inf*, 33, 1996.

[38] H. Völzer and D. Varacca. Defining fairness in reactive and concurrent systems. *J. ACM*, 59(3), 2012.